Algorithm Design Techniques

Over the last forty years, computer scientists have discovered that many algorithms share similar ideas, even though they solve very different problems. There appear to be relatively few basic techniques that can be applied when designing an algorithm, and we cover some of them later in this book in varying degrees of detail. For now we will mention the most common algorithm design techniques, so that future examples can be categorized in terms of the algorithm's design methodology.

To illustrate the design techniques, we will consider a very simple problem that plagues nearly everyone with a cordless phone. Suppose your cordless phone rings, but you have misplaced the handset somewhere in your home.

How do you find it? To complicate matters, you have just walked into your home with an armful of groceries, and it is dark out, so you cannot rely solely on eyesight.

Exhaustive Search

An *exhaustive search*, or *brute force*, algorithm examines every possible alternative to find one particular solution.

For example, if you used the brute force algorithm to find the ringing telephone, you would ignore the ringing of the phone, as if you could not hear it, and simply walk over every square inch of your home checking to see if the phone was present. You probably would not be able to answer the phone before it stopped ringing, unless you were very lucky, but you would be guaranteed to eventually find the phone no matter where it was.

BRUTEFORCECHANGE is a brute force algorithm, and chapter 4 introduces some additional examples of such algorithms—these are the easiest algorithms to design and understand, and sometimes they work acceptably for certain practical problems in biology. In general, though, brute force algorithms are too slow to be practical for anything but the smallest instances

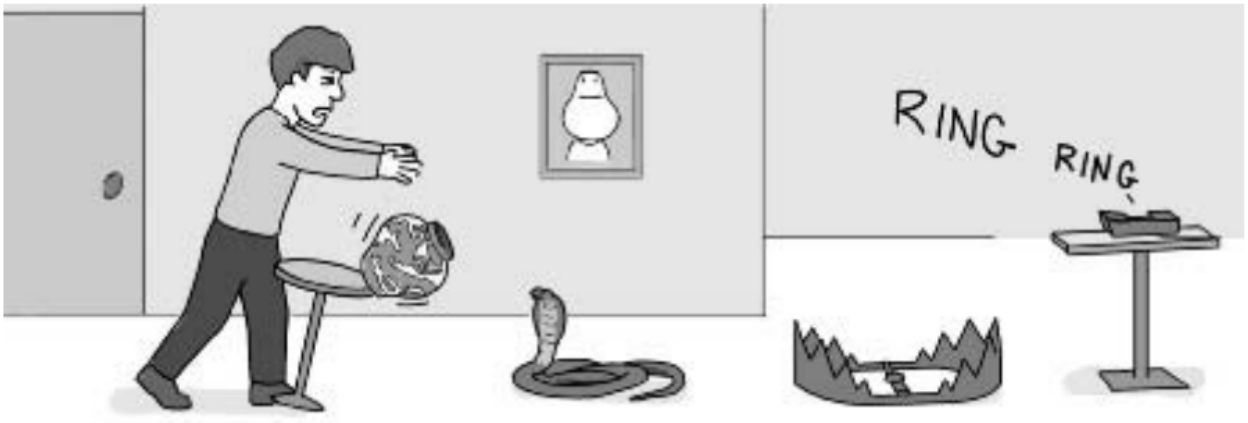
and we will spend most of this book demonstrating how to avoid the brute force algorithms or how to finesse them into faster versions.

Branch-and-Bound Algorithms

In certain cases, as we explore the various alternatives in a brute force algorithm, we discover that we can omit a large number of alternatives, a technique that is often called *branch-and-bound*, or *pruning*.

Suppose you were exhaustively searching the first floor and heard the phone ringing above your head. You could immediately rule out the need to search the basement or the first floor. What may have taken three hours may now only take one, depending on the amount of space that you can rule out.

Greedy Algorithms



Many algorithms are iterative procedures that choose among a number of alternatives at each iteration. For example, a cashier can view the Change problem as a series of decisions he or she has to make: which coin (among d denominations) to return first, which to return second, and so on. Some of these alternatives may lead to correct solutions while others may not. Greedy algorithms choose the "most attractive" alternative at each iteration, for example, the largest denomination possible. USCHANGE used quarters, then dimes, then nickels, and finally pennies (in that order) to make change for M. By greedily choosing the largest denomination first, the algorithm avoided any combination of coins that included fewer than three quarters to make change for an amount larger than or equal to 75 cents. Of course, we showed that the generalization of this greedy strategy, BETTERCHANGE, produced incorrect results when certain new denominations were included.

In the telephone example, the corresponding greedy algorithm would simply be to walk in the direction of the telephone's ringing until you found it. The problem here is that there may be a wall (or an expensive and fragile vase) between you and the phone, preventing you from finding it. Unfortunately, these sorts of difficulties frequently occur in most realistic problems. In many cases, a greedy approach will seem "obvious" and natural, but will be subtly wrong.

Dynamic Programming

Some algorithms break a problem into smaller subproblems and use the solutions of the subproblems to construct the solution of the larger one. During this process, the number of subproblems may become very large, and some algorithms solve the same subproblem repeatedly, needlessly increasing the running time. Dynamic programming organizes computations to avoid recomputing values that you already know, which can often save a great deal of time. The Ringing Telephone problem does not lend itself to a dynamic programming solution, so we consider a different problem to illustrate the technique.

Suppose that instead of answering the phone you decide to play the "Rocks" game from the previous chapter with two piles of rocks, say ten in each. We remind the reader that in each turn, one player may take either one rock (from either pile) or two rocks (one from each pile). Once the rocks are taken, they are removed from play. The player that takes the last rock wins the game. You make the first move.

To find the winning strategy for the $10+10$ game, we can construct a table, which we can call $\mathbf{R}$, shown below. Instead of solving a problem with 10 rocks in each pile, we will solve a more general problem with n rocks in one pile and m rocks in another (the $n+m$ game) where n and m are arbitrary. If Player 1 can always win the game of $5+6$, then we would say $R_{5,6}=W$, but if Player 1 has no winning strategy against a player that always makes the right moves, we would write $R_{5,6}=L$. Computing $R_{n,m}$ for an arbitrary n and m seems difficult, but we can build on smaller values. Some games, notably $R_{0,1}$, $R_{1,0}$, and $R_{1,1}$, are clearly winning propositions for Player 1 since in the first move, Player 1 can win. Thus, we fill in entries $(1,1)$, $(0,1)$ and $(1,0)$ as W.

	0	1	2	3	4	5	6	7	8	9	10
0		W									
1	W	W									
2											
3											
4											
5											
6											
7											
8											
9											
10											

After the entries $(0,1)$, $(1,0)$, and $(1,1)$ are filled, one can try to fill other entries. For example, in the $(2,0)$ case, the only move that Player 1 can make leads to the $(1,0)$ case that, as we already know, is a winning position for his opponent. A similar analysis applies to the $(0,2)$ case, leading to the following result:

	0	1	2	3	4	5	6	7	8	9	10
0		W	L								
1	W	W									
2	L										
3											
4											
5											
6											
7											
8											
9											
10											

In the $(2,1)$ case, Player 1 can make three different moves that lead respec-

tively to the games of $(1,1)$, $(2,0)$, or $(1,0)$. One of these cases, $(2,0)$, leads to a losing position for his opponent and therefore $(2,1)$ is a winning position. The case $(1,2)$ is symmetric to $(2,1)$, so we have the following table:

	0	1	2	3	4	5	6	7	8	9	10
0		W	L								
1	W	W	W								
2	L	W									
3											
4											
5											
6											
7											
8											
9											
10											

Now we can fill in $R_{2,2}$. In the $(2,2)$ case, Player 1 can make three different moves that lead to entries $(2,1)$, $(1,2)$, and $(1,1)$. All of these entries are winning positions for his opponent and therefore $R_{2,2} = L$

	0	1	2	3	4	5	6	7	8	9	10
0		W	L								
1	W	W	W								
2	L	W	L								
3											
4											
5											
6											
7											
8											
9											
10											

We can proceed filling in $\mathbf{R}$ in this way by noticing that for the entry (i,j) to be L, the entries above, diagonally to the left and directly to the left, must be W. These entries ($(i-1,j)$, $(i-1,j-1)$, and $(i,j-1)$) correspond to the three possible moves that player 1 can make.

	0	1	2	3	4	5	6	7	8	9	10
0		W	L	W	L	W	L	W	L	W	L
1	W	W	W	W	W	W	W	W	W	W	W
2	L	W	L	W	L	W	L	W	L	W	L
3	W	W	W	W	W	W	W	W	W	W	W
4	L	W	L	W	L	W	L	W	L	W	L
5	W	W	W	W	W	W	W	W	W	W	W
6	L	W	L	W	L	W	L	W	L	W	L
7	W	W	W	W	W	W	W	W	W	W	W
8	L	W	L	W	L	W	L	W	L	W	L
9	W	W	W	W	W	W	W	W	W	W	W
10	L	W	L	W	L	W	L	W	L	W	L

The ROCKS algorithm determines if Player 1 wins or loses. If Player 1 wins in an n+m game, ROCKS returns W. If Player 1 loses, ROCKS returns L. The ROCKS algorithm introduces an artificial initial condition, $R_{0,0} = L$ to simplify the pseudocode.

```
ROCKS(n, m)
 1  R_{0,0} = L
 2  for i ← 1 to n
 3       if R_{i-1,0} = W
 4            R_{i,0} ← L
 5       else
 6            R_{i,0} ← W
 7  for j ← 1 to m
 8       if R_{0,j-1} = W
 9            R_{0,j} ← L
10       else
11            R_{0,j} ← W
12  for i ← 1 to n
13       for j ← 1 to m
14            if R_{i-1,j-1} = W and R_{i,j-1} = W and R_{i-1,j} = W
15                 R_{i,j} ← L
16            else
17                 R_{i,j} ← W
18  return R_{n,m}
```

In point of fact, a faster algorithm to solve the Rocks puzzle relies on the simply pattern in $\mathbf{R}$, and checks to see if n and m are both even, in which

case the player loses.

```
FASTROCKS(n, m)
1  if  n and m are both even
2        return L
3  else
4        return W
```

However, though FASTROCKS is more efficient than ROCKS, it may be difficult to modify it for other games, for example a game in which each player can move up to three rocks at a time from the piles. This is one example where the slower algorithm is more instructive than a faster one. But obviously, it is usually better to use the faster one when you really need to solve the problem.

Divide-and-Conquer Algorithms

One big problem may be hard to solve, but two problems that are half the size may be significantly easier. In these cases, *divide-and-conquer* algorithms fare well by doing just that: splitting the problem into smaller subproblems, solving the subproblems independently, and combining the solutions of subproblems into a solution of the original problem. The situation is usually more complicated than this and after splitting one problem into subproblems, a divide-and-conquer algorithm usually splits these subproblems into even smaller sub-subproblems, and so on, until it reaches a point at which it no longer needs to recurse. A critical step in many divide-and-conquer algorithms is the recombining of solutions to subproblems into a solution for a larger problem. Often, this merging step can consume a considerable amount of time. We will see examples of this technique in chapter 7.

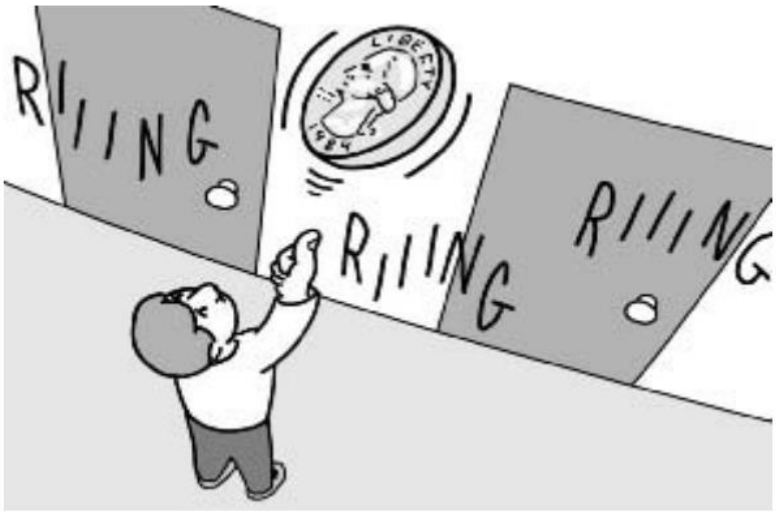
Machine Learning

Another approach to the phone search problem is to collect statistics over the course of a year about where you leave the phone, learning where the phone tends to end up most of the time. If the phone was left in the bathroom 80% of the time, in the bedroom 15% of the time, and in the kitchen 5% of the time, then a sensible time-saving strategy would be to start the search in

the bathroom, continue to the bedroom, and finish in the kitchen. Machine learning algorithms often base their strategies on the computational analysis of previously collected data.

Randomized Algorithms

If you happen to have a coin, then before even starting to search for the phone, you could toss it to decide whether you want to start your search on the first floor if the coin comes up heads, or on the second floor if the coin comes up tails. If you also happen to have a die, then after deciding on the second floor, you could roll it to decide in which of the six rooms on the second floor to start your search.[17] Although tossing coins and rolling dice may be a fun way to search for the phone, it is certainly not the intuitive thing to do, nor is it at all clear whether it gives you any algorithmic advantage over a deterministic algorithm. We will learn how randomized algorithms



help solve practical problems, and why some of them have a competitive advantage over deterministic algorithms.

Tractable versus Intractable Problems

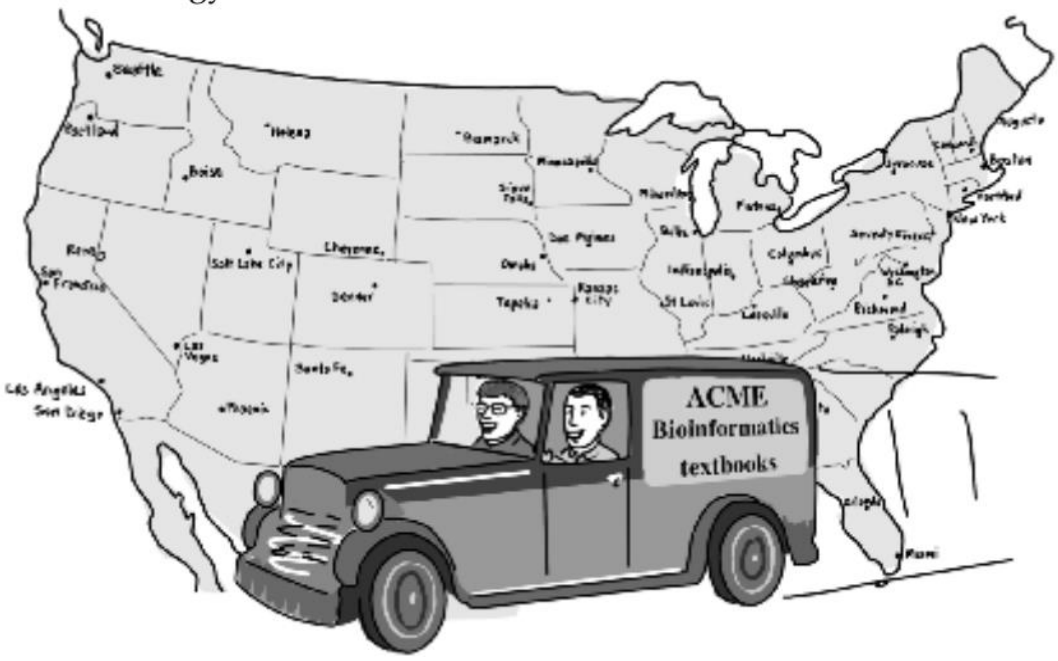
We have described a correct algorithm that solves the Change problem, but requires exponential time to do so. This does not mean that all algorithms that solve the Change problem will require exponential time. Showing that a

particular algorithm requires exponential time is much different than showing that a *problem* cannot be solved by *any* algorithm in less than exponential time. For example, we showed that RECURSIVEFIBONACCI required exponential time to compute the nth Fibonacci number, while FIBONACCI solves the same problem in linear $O(n)$ time.[18]

We have seen that algorithms can be categorized according to their complexity. In the early 1970s, computer scientists discovered that *problems* could also be categorized according to their inherent complexity. It turns out that some problems, such as listing every subset of an n-element set, require exponential time—no algorithm, no matter how clever, could possibly solve the problem in less than exponential time. Other problems, such as sorting a list of integers, require only polynomial time. Somewhere between the polynomial problems and the exponential problems lies one particularly important category of problems called the $\mathcal{NP}$*-complete* problems. These are problems

18. There exists an algorithm even faster than $O(n)$ to compute the n-th Fibonacci number; it does not calculate all of the Fibonacci numbers in the sequence up to n.

that appear to be quite difficult, in that no polynomial-time algorithm for any of these problems has yet been found. However, nobody can seem to prove that polynomial-time algorithms for these problems are impossible, so nobody can rule out the possibility that these problems are actually efficiently solvable. One particularly famous example of an $\mathcal{NP}$-complete problem is the Traveling Salesman problem, which has a wide variety of practical applications in biology.



Traveling Salesman Problem:
Find the shortest path through a set of cities, visiting each city only one time.

Input: A map of cities, roads between the cities, and distances along the roads.

Output: A sequence of roads that will take a salesman through every city on the map, such that he will visit each city exactly once, and will travel the shortest total distance.

The critical property of $\mathcal{NP}$-complete problems is that, if one $\mathcal{NP}$-complete problem is solvable by a polynomial-time algorithm, then *all* $\mathcal{NP}$-complete problems can be solved by minor modifications of the same algorithm. The

Greedy Algorithms

The algorithm USCHANGE in chapter 2 is an example of a greedy strategy: at each step, the cashier would only consider the largest denomination smaller than (or equal to) M. Since the goal was to minimize the number of coins returned to the customer, this seemed like a sensible strategy: you would never use five nickels in place of one quarter. A generalization of USCHANGE, BETTERCHANGE also used what seemed like the best option and did not consider any others, which is what makes an algorithm "greedy." Unfortunately, BETTERCHANGE actually returned incorrect results in some cases because of its short-sighted notion of "good." This is a common characteristic of greedy algorithms: they often return suboptimal results, but take very little time to do so. However, there are a lucky few greedy algorithms that find optimal rather than suboptimal solutions.

Genome Rearrangements

Waardenburg's syndrome is a genetic disorder resulting in hearing loss and pigmentary abnormalities, such as two differently colored eyes. The disease was named after the Dutch ophthalmologist who first noticed that people with two differently colored eyes frequently had hearing problems as well.

In the early 1990s, biologists narrowed the search for the gene implicated in Waardenburg's syndrome to human chromosome 2, but its exact location remained unknown for some time. There was another clue that shed light on the gene associated with Waardenburg's syndrome, that drew attention to chromosome 2: for a long time, breeders scrutinized mice for mutants, and one of these, designated *splotch*, had pigmentary abnormalities like patches of white spots, similar to those in humans with Waardenburg's syndrome. Through breeding, the *splotch* gene was mapped to one of the mouse chro-

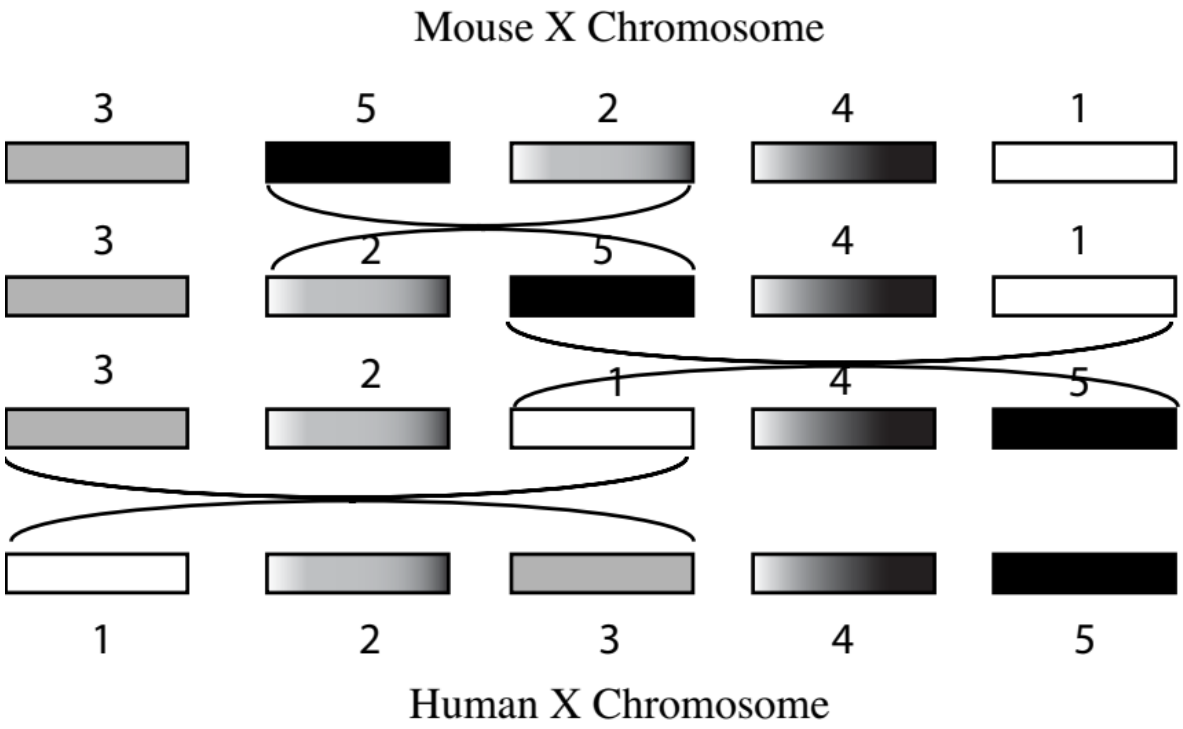


Figure 5.1 Transformation of the mouse gene order into the human gene order on the X chromosome (only the five longest synteny blocks are shown here).

mosomes. As gene mapping proceeded it became clear that there are groups of genes in mice that appear in the same order as they do in humans: these genes are likely to be present in the same order in a common ancestor of humans and mice—the ancient mammalian genome. In some ways, the human genome is just the mouse genome cut into about 300 large genomic fragments, called *synteny blocks*, that have been pasted together in a different order. Both sequences are just two different shufflings of the ancient mam-

malian genome. For example, chromosome 2 in humans is built from fragments that are similar to mouse DNA residing on chromosomes 1, 2, 3, 5, 6, 7, 10, 11, 12, 14, and 17. It is no surprise, then, that finding a gene in mice often leads to clues about the location of the related gene in humans.

Every genome rearrangement results in a change of gene ordering, and a series of these rearrangements can alter the genomic architecture of a species. Analyzing the rearrangement history of mammalian genomes is a challenging problem, even though a recent analysis of human and mouse genomes implies that fewer than 250 genomic rearrangements have occurred since the divergence of humans and mice approximately 80 million years ago. Every study of genome rearrangements involves solving the combinatorial puzzle of finding a series of rearrangements that transform one genome into another. Figure 5.1 presents a *rearrangement scenario* in which the mouse X chromosome is transformed into the human X chromosome.[1] The elementary rearrangement event in this scenario is the flipping of a genomic segment, called a *reversal*, or an *inversion*. One can consider other types of evolutionary events but in this book we only consider reversals, the most common evolutionary events.

Biologists are interested in the most parsimonious evolutionary scenario, that is, the scenario involving the smallest number of reversals. While there is no guarantee that this scenario represents an actual evolutionary sequence, it gives us a lower bound on the number of rearrangements that have occurred and indicates the similarity between two species.[2]

Even for the small number of synteny blocks shown, it is not so easy to verify that the three evolutionary events in figure 5.1 represent a *shortest* series of reversals transforming the mouse gene order into the human gene order on the X chromosome. The exhaustive search technique that we presented in the previous chapter would hardly work for rearrangement studies since the number of variants that need to be explored becomes enormous for more than ten synteny blocks. Below, we explore two greedy approaches that work to differing degrees of success.

Sorting by Reversals

In their simplest form, rearrangement events can be modeled by a series of reversals that transform one genome into another. The order of genes (rather, of synteny blocks) in a genome can be represented by a permutation[3]

1. Extreme conservation of genes on X chromosomes across mammalian species provides an opportunity to study the evolutionary history of X chromosomes independently of the rest of the genomes, since the gene content of X chromosomes has barely changed throughout mammalian evolution. However, the order of genes on X chromosomes has been disrupted several times. In other words, genes that reside on the X chromosome stay on the X chromosome (but their order may change). All other chromosomes may exchange genes, that is, a gene can move from one chromosome to another.
2. In fact, a sequence of reversals that transforms the X chromosome of mouse into the X chromosome of man does not even represent an evolutionary sequence, since humans are not descended from the present-day mouse. However, biologists believe that the architecture of the X chromosome in the human-mouse ancestor is about the same as the architecture of the human X chromosome.
3. A permutation of a sequence of n numbers is just a reordering of that sequence. We will always use permutations of consecutive integers: for example, $2\,1\,3\,4\,5$ is a permutation of $1\,2\,3\,4\,5$.

$\pi = \pi_1\pi_2\cdots\pi_n$. The order of synteny blocks on the X chromosome in humans is represented in figure 5.1 by $(1, 2, 3, 4, 5)$, while the ordering in mice is $(3, 5, 2, 4, 1)$.[4]

A reversal $\rho(i, j)$ has the effect of reversing the order of synteny blocks

$$\pi_i\pi_{i+1}\cdots\pi_{j-1}\pi_j$$

In effect, this transforms

$$\pi = \pi_1\cdots\pi_{i-1}\underrightarrow{\pi_i\pi_{i+1}\cdots\pi_{j-1}\pi_j}\pi_{j+1}\cdots\pi_n$$

into

$$\pi\cdot\rho(i,j) = \pi_1\cdots\pi_{i-1}\underleftarrow{\pi_j\pi_{j-1}\cdots\pi_{i+1}\pi_i}\pi_{j+1}\cdots\pi_n$$

For example, if $\pi = 1\,2\,\underrightarrow{4\,3\,7\,5}\,6$, then $\pi\cdot\rho(3,6) = 1\,2\,\underleftarrow{5\,7\,3\,4}\,6$. With this representation of a genome, and a rigorous definition of an evolutionary event, we are in a position to formulate the computational problem that mimics the biological rearrangement process.

Reversal Distance Problem:
Given two permutations, find a shortest series of reversals that transforms one permutation into another.

Input: Permutations π and σ.

Output: A series of reversals $\rho_1, \rho_2, \ldots, \rho_t$ transforming π into σ (i.e., $\pi \cdot \rho_1 \cdot \rho_2 \cdots \rho_t = \sigma$), such that t is minimum.

We call t the *reversal distance between* π *and* σ, and write $d(\pi, \sigma)$ to denote the reversal distance for a given π and σ. In practice, one usually selects the second genome's order as a gold standard, and arbitrarily sets σ to be the *identity permutation* $1\,2 \cdots n$. The Sorting by Reversals problem is similar to the Reversal Distance problem, except that it requires only one permutation as input.

4. In reality, genes and synteny blocks have directionality, reflecting whether they reside on the direct strand or the reverse complement strand of the DNA. In other words, the synteny block order in an organism is really represented by a *signed* permutation. However, in this section we

---

Sorting by Reversals Problem:
Given a permutation, find a shortest series of reversals that transforms it into the identity permutation.

Input: Permutation π.

Output: A series of reversals $\rho_1, \rho_2, \ldots, \rho_t$ transforming π into the identity permutation such that t is minimum.

In this case, we call t the *reversal distance of* π and denote it as $d(\pi)$. When sorting a permutation $\pi = 1\,2\,3\,6\,4\,5$, it hardly makes sense to move the already-sorted first three elements of π. If we define *prefix*(π) to be the number of already-sorted elements of π, then a sensible strategy for sorting by reversals is to increase *prefix*(π) at every step. This approach sorts π in 2 steps: $1\,2\,3\underline{6\,4}\,5 \rightarrow 1\,2\,3\,4\underline{6\,5} \rightarrow 1\,2\,3\,4\,5\,6$. Generalizing this leads to an algorithm that sorts a permutation by repeatedly moving its ith element to the ith position.[5]

```
SIMPLEREVERSALSORT(π)
1  for i ← 1 to n − 1
2      j ← position of element i in π (i.e., π_j = i)
3      if j ≠ i
4          π ← π · ρ(i, j)
5          output π
6      if π is the identity permutation
7          return
```

SIMPLEREVERSALSORT is an example of a greedy algorithm that chooses the "best" reversal at every step. However, the notion of "best" here is rather short-sighted—simply increasing $prefix(\pi)$ does not guarantee the smallest number of reversals. For example, SIMPLEREVERSALSORT takes five steps to sort 6 1 2 3 4 5:

$$\underline{6\,1}\,2\,3\,4\,5 \rightarrow 1\,\underline{6\,2}\,3\,4\,5 \rightarrow 1\,2\,\underline{6\,3}\,4\,5 \rightarrow 1\,2\,3\,\underline{6\,4}\,5 \rightarrow 1\,2\,3\,4\,\underline{6\,5} \rightarrow 1\,2\,3\,4\,5\,6$$

However, the same permutation can be sorted in just two steps:

$$\underline{6\,1\,2\,3\,4\,5} \rightarrow \underline{5\,4\,3\,2\,1}\,6 \rightarrow 1\,2\,3\,4\,5\,6.$$

Therefore, we can confidently say that SIMPLEREVERSALSORT is not a correct algorithm, in the strict sense of chapter 2. In fact, despite its commonsense appeal, SIMPLEREVERSALSORT is a terrible algorithm since it takes $n-1$ steps to sort the permutation $\pi = n\,1\,2\,\ldots\,(n-1)$ even though $d(\pi) = 2$.

Even before biologists faced genome rearrangement problems, computer scientists studied the related Sorting by Prefix Reversals problem, also known as the Pancake Flipping problem: given an arbitrary permutation π, find $d_{pref}(\pi)$, which is the minimum number of reversals of the form $\rho(1, i)$ sorting π. The Pancake Flipping problem was inspired by the following "real-life" situation described by (the fictitious) Harry Dweighter:

> The chef in our place is sloppy, and when he prepares a stack of pancakes they come out all different sizes. Therefore, when I deliver them to a customer, on the way to a table I rearrange them (so that the smallest winds up on top, and so on, down to the largest at the bottom) by grabbing several from the top and flipping them over, repeating this (varying the number I flip) as many times as necessary. If there are n pancakes, what is the maximum number of flips that I will ever have to use to rearrange them?

An analog of SIMPLEREVERSALSORT will sort every permutation by at most $2(n-1)$ prefix reversals. For example, one can sort $1\,2\,3\,6\,4\,5$ by 4 prefix reversals ($\underline{1\,2\,3\,6}\,4\,5 \rightarrow \underline{6\,3\,2\,1\,4\,5} \rightarrow \underline{5\,4\,1\,2\,3}\,6 \rightarrow \underline{3\,2\,1}\,4\,5\,6 \rightarrow 1\,2\,3\,4\,5\,6$) but it is not clear whether there exists an even shorter series of prefix reversals to sort this permutation. William Gates, an undergraduate student at Harvard in the mid-1970s, and Christos Papadimitriou, a professor at Harvard in the mid-1970s, now at Berkeley, made the first attempt to solve this problem and proved that any permutation can be sorted by at most $\frac{5}{3}(n+1)$ prefix reversals. However, the Pancake Flipping problem remains unsolved.